Sp17 Midterm Q1

1. Integers and Floats (7 points)

a. In the card game Schnapsen, 5 cards are used (Ace, Ten, King, Queen, and Jack) from 4 suits, so 20 cards in total. What are the minimum number of bits needed to represent a single card in a Schnapsen deck?

5

We need 2 bits to represent 4 suits, and 3 bits to represent 5 ranks. So 5 bits in total.

b. How many negative numbers can we represent if given 7 bits and using two's complement?

$\mathbf{2^6}$

Using 7 bits, the MSB has to be 1 for negative numbers. So there are 2^6 negative numbers in total.

Consider the following pseudocode (we've written out the bits instead of listing hex digits):

```
int a = 0b0100 0000 0000 0000 0000 0011 1100 0000
int b = (int)(float)a
int m = 0b0100 0000 0000 0000 0000 0011 0000 0000
int n = (int)(float)m
```

c. Circle one: True or [✓ **False**]:

`a == b`

The right-most 1 will be truncated (cannot fit in Mantissa)

d. Circle one: [✓ **True**] or False:

`m == n`

No precision will be lost

e. How many IEEE single precision floating point numbers are in the range [4, 6) (That is, how many floating point numbers are there where 4 <= x < 6?)

$\mathbf{2^{22}}$

4 in binary is $1.0 \cdot 2^2$.

6 in binary is $1.1 \cdot 2^2$.

So in Mantissa the right-most 22 bits can be either 0 or 1. Therefore, there are 2^{22} bits in range $[4, 6)$

Au15 Midterm Q5

5. Stack Discipline (30 points)

Examine the following recursive function:

```
long magic(long x, long *y) {
  long temp;
  if (x < 2) {
    return *y;
  } else {
    temp = *y + 1;
    return x + magic(x-3, &temp);
  }
}
```

Here is the x86_64 assembly for the same function:

```
4005f6 <magic>:
4005f6:   cmp    $0x1,%rdi
4005fa:   jg     0x400600 <magic+10>
4005fc:   mov    (%rsi),%rax
4005ff:   retq
400600:   push   %rbx
400601:   sub    $0x10,%rsp
400605:   mov    %rdi,%rbx
400608:   mov    (%rsi),%rax
40060b:   add    $0x1,%rax
40060f:   mov    %rax,0x8(%rsp)
400614:   lea    -0x3(%rdi),%rdi
400618:   lea    0x8(%rsp),%rsi
40061d:   callq  0x4005f6 <magic>
400622:   add    %rbx,%rax
400625:   add    $0x10,%rsp
400629:   pop    %rbx
40062a:   retq
```

Suppose we call **`magic`** from **`main()`**, with registers %**`rsi`** = **`0x7ff…ffbaa`** and %**`rdi`** = 7. The value stored at address **`0x7ff…ffbaa`** is the long value 3. We set a breakpoint at "**`return *y`**" (i.e. we are just about to return from **`magic()`** without making another recursive call). We have executed the **`mov`** instruction at **`4005fc`** but have not yet executed the **`retq`**.

Fill in the register values on the next page and **draw what the stack will look like when the program hits that breakpoint**. Give both a description of the item stored at that location and the value stored at that location. If a location on the stack is not used, write "unused" in the Description for that address and put "-----" for its Value. You may list the Values in hex or decimal. Unless preceded by **`0x`** we will assume decimal. It is fine to use **`f…f`** for sequences of **`f`**'s as shown above for %**`rsi`**. Add more rows to the table as needed. Also, fill in the box on the next page to include the value this call to **`magic`** will *finally* return to **`main`**.

Version 1

Register	Original Value	Value at Breakpoint
`rsp`	`0x7ff…ffad0`	`0x7fffffffffffa90`
`rdi`	7	1
`rsi`	`0x7ff…ffbaa`	`0x7fffffffffffaa0`
`rbx`	2	4
`rax`	9	5

DON'T FORGET → What value is **finally** returned to **`main`** by this call? 16

Memory address on stack	Name/description of item	Value
0x7fffffffffffad0	Return address back to `main`	0x400827
0x7fffffffffffac8	Old rbx	2
0x7fffffffffffac0	temp	4
0x7fffffffffffab8	Unused	--------
0x7fffffffffffab0	Return address	0x400622
0x7fffffffffffaa8	Old rbx	7
0x7fffffffffffaa0	temp	5
0x7fffffffffffa98	Unused	--------
0x7fffffffffffa90	Return address	0x400622
0x7fffffffffffa88		
0x7fffffffffffa80		
0x7fffffffffffa78		
0x7fffffffffffa70		
0x7fffffffffffa68		
0x7fffffffffffa60		

Wi17 Final Q1

1. C and Assembly (15 points)

Consider the following (partially blank) x86-64 assembly, (partially blank) C code, and memory listing. Addresses and values are 64-bit, and the machine is little-endian. All the values in memory are in hex, and the address of each cell is the sum of the row and column headers: for example, address `0x1019` contains the value `0x18`.

Assembly code:

```
foo:
  movl $0, %eax

L1:
  cmpq 0x0, %rdi
  je L2
  cmp 0x18, 0x1(%rdi)
  je L3
  mov 0x8(%rdi), %rdi
  jmp L1

L2:
  ret

L3:
  mov (%rdi), %eax
  jmp L2
```

C code:

```
typedef struct person {
  char height;
  char age;
  struct person* next_person;
} person;

int foo(person* p) {
    int answer = 0;
    while (p != NULL) {
        if (p->age == 24){
          answer = p->height;
          break;
        }
        p = p->next_person;
    }
    return answer;
}
```

Memory Listing
Bits not shown are 0.

	0x00	0x01	...	0x05	0x06	0x07
0x1000	80	1B	...	00	00	00
0x1008	80	1B	...	00	00	00
0x1010	3F	18	...	00	00	00
0x1018	3F	18	...	00	00	00
0x1020	00	00	...	00	00	00
0x1028	18	10	...	00	00	00
0x1030	18	10	...	00	00	00
0x1038	40	40	...	00	00	00
0x1040	40	40	...	00	00	00
0x1048	00	00	...	00	00	00

(a) Given the code provided, fill in the blanks in the C and assembly code.

(b) Trace the execution of the call to `foo((person*) 0x1028)` in the table to the right. Show which instruction is executed in each step until `foo` returns. In each space, place **the assembly instruction** and the values of the appropriate registers **after that instruction executes**. *You may leave those spots blank when the value does not change.* You might not need all steps listed on the table.

Instruction	`%rdi` (hex)	`%eax` (decimal)
`movl`	`0x1028`	0
`cmpq`		
`je`		
`cmp`		
`je`		
`mov`	`0x1018`	
`jmp`		
`cmpq`		
`je`		
`cmp`		
`je`		
`mov`		63
`jmp`		
`ret`		

(c) Briefly describe the value that `foo` returns and how it is computed. Use only variable names from the C version in your answer.

foo traverses a linked list of person structs, and returns the height of the first person whose age == 24.

Sp17 Final Q1

1. Caches (11 points)

You are using a byte-addressed machine where physical addresses are 22-bits. You have a 4-way associative cache of total size 1 KiB with a cache block size of 32 bytes. It uses LRU replacement and write-back policies.

a) Give the number of bits needed for each of these:

Cache Block Offset: ______**5**______ Cache Tag: ______**14**________

b) How many sets will the cache have? _____**8**______

c) Assume that everything except the array **x** is stored in registers, and that the array **x** starts at address 0x0. Give the hit rate (as a fraction or a %) for the following code, assuming that the cache starts out empty. Also give the total number of hits.

```
#define LEAP 1
#define SIZE 256
int x[SIZE][8];
... // Assume x has been initialized to contain values.
... // Assume the cache starts empty at this point.
for (int i = 0; i < SIZE; i += LEAP) {
  x[i][0] += x[i][4];
}
```

Hit Rate: ___**2/3**_______ Total Number of **Hits**: _____**512**______

d) If we increase the cache block size to 64 bytes (and leave all other factors the same) what would the hit rate be?

Hit Rate: ______**5/6**_______ Total Number of **Hits**: ______**640**_______

e) For each of the changes proposed below, indicate how it would affect the **hit rate** of the code above in part c) *assuming that all other factors remained the same* as they were in the original cache:

Change associativity from 4-way to 2-way:	increase	/	[✓ no change]	/	decrease
Change **LEAP** from 1 to 4:	increase	/	[✓ no change]	/	decrease
Change cache size from 1 KiB to 2 KiB:	increase	/	[✓ no change]	/	decrease

Name:____

Wi16 Final Q4

4. *Processes* (**12** points) In this problem, assume Linux.

   (a) Can the same program be executing in more than one process simultaneously?

   (b) Can a single process change what program it is executing?

   (c) When the operating system performs a context switch, what information does *NOT* need to be saved/maintained in order to resume the process being stopped later (circle all that apply):

   - The page-table base register
   - The value of the stack pointer
   - The time of day (i.e., value of the clock)
   - The contents of the TLB
   - The process-id
   - The values of the process' global variables

   (d) Give an example of an exception (asynchronous control flow) in which it makes sense to later re-execute the instruction that caused the exception.

   (e) Give an example of an exception (asynchronous control flow) in which it makes sense to abort the process.

Solution:

(a) Yes (the question is ambiguous as to what "simultaneous" means. We clarified during the exam, "Assume it *is* the case that multiple processes execute simultaneously. Then the question is whether more than one of these processes can be executing the same program." Under this interpretation, only "yes" is plausibly correct.)

(b) Yes

(c) The time of day and the contents of the TLB

(d) Page fault for memory on disk (other answers possible; full credit given just for page-fault even though that's ambiguous)

(e) Division by zero (other answers possible)

Name: ____________________________________

Sp16 Final Q6

6. Programs, processes, and processors (oh my!) (25 pts)

(a) Consider the following C code on the left (running on Linux), then give *one* possible output of running it. Assume that `printf` flushes its output immediately.

```
void oz() {
    char * name = "toto\n";
    printf("dorothy\n");
    if (fork() == 0) {
        name = "wizard\n";
        printf("scarecrow\n");
        fork();
        printf("tinman\n");
        exit(0);
        printf("witch\n");
    } else {
        printf("lion\n");
    }
    printf(name);
}
```

Possible output:

```
dorothy          dorothy
scarecrow        lion
tinman           toto
tinman           scarecrow
lion             tinman
toto             tinman
```

(b) *"Pay no attention to the man behind the curtain."* We have seen several different mechanisms used to create illusions or abstractions for running programs:

A. Context switch

B. Virtual memory

C. Virtual method tables (vtables)

D. Caches

E. Timer interrupt

F. Stack discipline

G. None of the above, or impossible.

For each of the following, indicate which mechanism above (**A**-**F**) enables the behavior, or **G** if the behavior is impossible or untrue.

(i) __E__ Allows operating system kernel to run to make scheduling decisions.

(ii) __G__ Prevents buffer overflow exploits.

(iii) __B__ Allows multiple instances of the same program to run concurrently.

(iv) __B__ Lets programs use more memory than the machine has.

(v) __D__ Makes recently accessed memory faster.

(vi) __A__ Multiple processes appear to run concurrently on a single processor.

(vii) __C__ Enables programs to run different code depending on an object's type.

(viii) __G__ Allows an x86-64 machine to execute code for a different ISA.

Name: ______________________________

(c) Give an example of a *synchronous* exception, what could trigger it, and where the exception handler would return control to in the original program.

Page fault: triggered by access to virtual address not in memory, returns to the instruction that caused the fault.

Trap: used to for syscalls to do something protected by the kernel, returns to after the calling instruction.

(d) In what way does address translation (virtual memory) help make `exec` fast? Explain in less than 2 sentences. *Hint:* it may help to write down what happens during `exec`.

Address translation is a form of *indirection*, it allows us to implement `fork` without copying the whole process's memory, and `exec` without loading the whole program into memory at once.

(e) Which of the following *can* a running process determine, assuming it does *not* have access to a timer? *(check all that apply)*

- [X] Its own process ID
- [ ] Size of physical memory
- [X] Size of the virtual address space
- [ ] L1 cache associativity
- [ ] When context switches happen

(f) For each of the following, fill in what is responsible for making the decision: hardware ("HW"), operating system ("OS"), or program ("P").

(i) __OS__ Which physical page a virtual page is mapped to.

(ii) __HW__ Which cache line is evicted for a conflict in a set-associative cache.

(iii) __OS__ Which page is evicted from physical memory during a page fault.

(iv) __HW__ Translation from virtual address to physical address.

(v) __P__ Whether data is stored in the stack or the heap.

(vi) __P__ Data layout optimized for spatial locality

Sp17 Final Q3

3. Virtual Memory (9 points)

Assume we have a virtual memory detailed as follows:

- 256 MiB Physical Address Space
- 4 GiB Virtual Address Space
- 1 KiB page size
- A TLB with 4 sets that is 8-way associative with LRU replacement

For the following questions it is fine to leave your answers as powers of 2.

a) How many bits will be used for:

Page offset? _____**10**______

Virtual Page Number (VPN)? ____**22**_____ Physical Page Number (PPN)? ___**18**______

TLB index? _______**2**_________ TLB tag? _______**20**__________

b) How many entries in this page table?

$$2^{22}$$

c) We run the following code with an empty TLB. Calculate the TLB miss rate for data (ignore instruction fetches). Assume `i` and `sum` are stored in registers and `cool` is page-aligned.

```
#define LEAP 8
int cool[512];
... // Some code that assigns values into the array cool
... // Now flush the TLB. Start counting TLB miss rate from here.
int sum;
for (int i = 0; i < 512; i += LEAP) {
  sum += cool[i];
}
```

TLB Miss Rate: (fine to leave you answer as a fraction) ____ $\frac{1}{32}$ _________

Au16 Final Q7

Question F7: Virtual Memory [10 pts]

Our system has the following setup:

- 24-bit virtual addresses and 512 KiB of RAM with 4 KiB pages
- A 4-entry TLB that is fully associative with LRU replacement
- A page table entry contains a valid bit and protection bits for read (R), write (W), execute (X)

(A) Compute the following values: [2 pt]

Page offset width	__**12**__	PPN width	__**7**__
Entries in a page table	__2^{12}__	TLBT width	__**12**__

Because TLB is fully associative, TLBT width matches VPN. There are $2^{\text{VPN width}}$ entries in PT.

(B) Briefly explain why we make the page size so much larger than a cache block size. [2 pt]

Take advantage of spatial locality and try to avoid page faults as much as possible. Disk access is also super slow, so we want to pull a lot of data when we do access it.

(C) Fill in the following blanks with "**A**" for always, "**S**" for sometimes, and "**N**" for never if the following get updated during a **page fault**. [2 pt]

Page table __**A**__ Swap space __**S**__ TLB _**A/N**_ Cache __**S**__

When the page is place in physical memory, the new PPN is written into the **page table** entry.
Swap space will get updated if a dirty page is kicked out of physical memory.
For this class, we say that the page fault handler updates the **TLB** because it is more efficient.
In reality not all do (OS does not have access to hardware-only TLB; instr gets restarted).
To update a PTE (in physical mem), you check the **cache**, so it gets updated on a cache miss.

(D) The TLB is in the state shown when the following code is executed. Which iteration (value of `i`) will cause the **protection fault (segfault)**? Assume `sum` is stored in a register.
Recall: the hex representations for TLBT/PPN are padded as necessary. [4 pt]

```
long *p = 0x7F0000, sum = 0;
for (int i = 0; 1; i++) {
   if (i%2)
      *p = 0;
   else
      sum += *p;
   p++;
}
```

TLBT	PPN	Valid	R	W	X
0x7F0	0x31	1	1	1	0
0x7F2	0x15	1	1	0	0
0x004	0x1D	1	1	0	1
0x7F1	0x2D	1	1	0	0

`i` = **513**

Only the current page (VPN = TLBT = 0x7F0) has write access. Once we hit the next page (TLBT = 0x7F1), we will encounter a segfault once we try to *write* to the page. We are using pointer arithmetic to increment our pointer by 8 bytes at a time. One page holds $2^{12}/2^3 = 512$ `long`s, so we first access TLBT 0x7F1 when `i` = 512. However, the code is set up so that we only write on *odd* values of `i`, so the answer is `i` = 513.

Au16 Final Q8

Question F8: Memory Allocation [9 pts]

(A) Briefly describe one drawback and one benefit to using an *implicit* free list over an *explicit* free list. [4 pt]

Implicit drawback:	Implicit benefit:
• Slower – have to check both allocated and free blocks • Must use both boundary tags in every block – less room for payload	• Simpler code; easier to manage • Smaller minimum block size (less internal fragmentation for free blocks)

(B) The table shown to the right shows the *value of the header* for the block returned by the request: **`(int*)malloc(N*sizeof(int))`** What is the alignment size for this dynamic memory allocator? [2 pt]

N	header value
6	33
8	49
10	49
12	65

16 bytes

The alignment size is given by the difference in size once we cross an alignment boundary. Remembering to mask out the allocated tag, we see that 6 `ints` = 24 bytes gets rounded up to 32 and 8 `ints` = 32 bytes gets rounded up to 48 (remember extra space for internal fragmentation – at least the header, possibly other things).

(C) Consider the C code shown here. Assume that the `malloc` call succeeds and `foo` is stored in memory (not just in a register). Fill in the following blanks with ">" or "<" to compare the *values* returned by the following expressions just before `return 0`. [3 pt]

```
#include <stdlib.h>
int ZERO = 0;
char* str = "cse351";

int main(int argc, char *argv[]) {
    int *foo = malloc(8);
    free(foo);
    return 0;
}
```

`ZERO`	__<__	`&ZERO`
`foo`	__<__	`&foo`
`foo`	__>__	`&str`

`ZERO` and `str` are global variables, so their *addresses* are in the Static Data section of memory.
`str`'s *value* is the address of a string literal, which sits at the bottom portion of Static Data.
`foo` is a local variable, so its *address* is in the Stack, but its *value* is the address of a block in the Heap.
The virtual address space is arranged such that 0 < Instructions < Static Data < Heap < Stack.

Name:______________________________________

Wi16 Final Q10

10. *C vs. Java* (**11** points) Consider this Java code (left) and somewhat similar C code (right) running on x86-64:

```
public class Foo {
  private int[] x;
  private int y;
  private int z;
  private Bar b;
  public Foo() {
     x = null;
     b = null;
  }
}
```

```
struct Foo {
  int x[6];
  int y;
  int z;
  struct Bar * b;
};

struct Foo * make_foo() {
  struct Foo * f = (struct Foo *)malloc(sizeof(struct Foo));
  f->x = NULL;
  f->b = NULL;
  return f;
}
```

(a) In Java, `new Foo()` allocates a new object on the heap. How many bytes would you expect this object to contain for holding `Foo`'s fields? (Do *not* include space for any header information, vtable pointers, or allocator data.)

(b) In C, `malloc(sizeof(struct Foo))` allocates a new object on the heap. How many bytes would you expect this object to contain for holding `struct Foo`'s fields? (Do *not* include space for any header information or allocator data.)

(c) The function `make_foo` attempts to be a C variant of the `Foo` constructor in Java. One line fails to compile. Which one and why?

(d) What, if anything, do we know about the values of the `y` and `z` fields after Java creates an instance of `Foo`?

(e) What, if anything, do we know about the values of the `y` and `z` fields in the object returned by `make_foo`?

Solution:

(a) 24

(b) 40

(c) `f->x = NULL` does not compile. In C, the field declaration `int x[6]` creates an inline array, not a pointer, so it does not make any sense to "assign NULL to the array" — the struct itself has slots for six array elements.

(d) We know both fields hold 0.

(e) We know nothing. (We know something abou their size, but not their contents – it could be any bit-pattern.)